****
*** 162617 Thu Jun 30 09:24:11 2016***
new/src/os/aix/vm/os_aix.cpp
*** NO COMMENTS ***
****
_____***unchanged_portion_omitted_***

```
1036 bool os::supports_vtime() { return true; }
1037 bool os::enable_vtime()   { return false; }
1038 bool os::vtime_enabled()  { return false; }
```

```
1036 double os::elapsedVTime() {
1037   struct rusage usage;
1038   int retval = getrusage(RUSAGE_THREAD, &usage);
1039   if (retval == 0) {
1040     return usage.ru_utime.tv_sec + usage.ru_stime.tv_sec + (usage.ru_utime.tv_us
1041   } else {
1042     // better than nothing, but not much
1043     return elapsedTime();
1044   }
1045 }
```

_____***unchanged_portion_omitted_***

****
147019 Thu Jun 30 09:24:11 2016
new/src/os/bsd/vm/os_bsd.cpp
****** NO COMMENTS ******
****
_____***unchanged_portion_omitted_***

```
 931 bool os::supports_vtime() { return true; }
 932 bool os::enable_vtime()   { return false; }
 933 bool os::vtime_enabled()  { return false; }
```

```
 931 double os::elapsedVTime() {
 932   // better than nothing, but not much
 933   return elapsedTime();
 934 }
```

_____***unchanged_portion_omitted_***

****
*** 213447 Thu Jun 30 09:24:11 2016***
new/src/os/linux/vm/os_linux.cpp
*** NO COMMENTS ***
****
_____***unchanged_portion_omitted_***

```
1169 bool os::supports_vtime() { return true; }
1170 bool os::enable_vtime()   { return false; }
1171 bool os::vtime_enabled()  { return false; }
```

```
1169 double os::elapsedVTime() {
1170   struct rusage usage;
1171   int retval = getrusage(RUSAGE_THREAD, &usage);
1172   if (retval == 0) {
1173     return (double) (usage.ru_utime.tv_sec + usage.ru_stime.tv_sec) + (double) (
1174   } else {
1175     // better than nothing, but not much
1176     return elapsedTime();
1177   }
1178 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  199544 Thu Jun 30 09:24:11 2016
new/src/os/solaris/vm/os_solaris.cpp
*** NO COMMENTS ***
**********************************************************
______unchanged_portion_omitted_

1322 bool os::supports_vtime() { return true; }

1324 bool os::enable_vtime() {
1325   int fd = ::open("/proc/self/ctl", O_WRONLY);
1326   if (fd == -1) {
1327     return false;
1328   }

1330   long cmd[] = { PCSET, PR_MSACCT };
1331   int res = ::write(fd, cmd, sizeof(long) * 2);
1332   ::close(fd);
1333   if (res != sizeof(long) * 2) {
1334     return false;
1335   }
1336   return true;
1337 }

1339 bool os::vtime_enabled() {
1340   int fd = ::open("/proc/self/status", O_RDONLY);
1341   if (fd == -1) {
1342     return false;
1343   }

1345   pstatus_t status;
1346   int res = os::read(fd, (void*) &status, sizeof(pstatus_t));
1347   ::close(fd);
1348   if (res != sizeof(pstatus_t)) {
1349     return false;
1350   }
1351   return status.pr_flags & PR_MSACCT;
1352 }

1322 double os::elapsedVTime() {
1323   return (double)gethrvtime() / (double)hrtime_hz;
1324 }
______unchanged_portion_omitted_
```

****
194985 Thu Jun 30 09:24:11 2016
new/src/os/windows/vm/os_windows.cpp
****** NO COMMENTS ******
****
_____***unchanged_portion_omitted_***

```
 878 bool os::supports_vtime() { return true; }
 879 bool os::enable_vtime() { return false; }
 880 bool os::vtime_enabled() { return false; }
```

```
 878 double os::elapsedVTime() {
 879   FILETIME created;
 880   FILETIME exited;
 881   FILETIME kernel;
 882   FILETIME user;
 883   if (GetThreadTimes(GetCurrentThread(), &created, &exited, &kernel, &user) != 0
 884     // the resolution of windows_to_java_time() should be sufficient (ms)
 885     return (double) (windows_to_java_time(kernel) + windows_to_java_time(user))
 886   } else {
 887     return elapsedTime();
 888   }
 889 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    6292 Thu Jun 30 09:24:12 2016
new/src/share/vm/gc/g1/concurrentG1RefineThread.cpp
*** NO COMMENTS ***
**********************************************************
______unchanged_portion_omitted_

 109 void ConcurrentG1RefineThread::run_service() {
 110   _vtime_start = os::elapsedVTime();

 112   while (!should_terminate()) {
 113     // Wait for work
 114     wait_for_completed_buffers();
 115     if (should_terminate()) {
 116       break;
 117     }

 119     size_t buffers_processed = 0;
 120     DirtyCardQueueSet& dcqs = JavaThread::dirty_card_queue_set();
 121     log_debug(gc, refine)("Activated %d, on threshold: " SIZE_FORMAT ", current:
 122                           _worker_id, _activation_threshold, dcqs.completed_buff

 124     {
 125       SuspendibleThreadSetJoiner sts_join;

 127       while (!should_terminate()) {
 128         if (sts_join.should_yield()) {
 129           sts_join.yield();
 130           continue;             // Re-check for termination after yield delay.
 131         }

 133         size_t curr_buffer_num = dcqs.completed_buffers_num();
 134         // If the number of the buffers falls down into the yellow zone,
 135         // that means that the transition period after the evacuation pause has
 136         if (dcqs.completed_queue_padding() > 0 && curr_buffer_num <= cg1r()->yel
 137           dcqs.set_completed_queue_padding(0);
 138         }

 140         // Check if we need to activate the next thread.
 141         if ((_next != NULL) &&
 142             !_next->is_active() &&
 143             (curr_buffer_num > _next->_activation_threshold)) {
 144           _next->activate();
 145         }

 147         // Process the next buffer, if there are enough left.
 148         if (!dcqs.apply_closure_to_completed_buffer(_refine_closure,
 149                                                     _worker_id + _worker_id_offs
 150                                                     _deactivation_threshold,
 151                                                     false /* during_pause */)) {
 152           break; // Deactivate, number of buffers fell below threshold.
 153         }
 154         ++buffers_processed;
 155       }
 156     }

 158     deactivate();
 159     log_debug(gc, refine)("Deactivated %d, off threshold: " SIZE_FORMAT
 160                           ", current: " SIZE_FORMAT ", processed: " SIZE_FORMAT,
 161                           _worker_id, _deactivation_threshold,
 162                           dcqs.completed_buffers_num(),
 163                           buffers_processed);

 165     if (os::supports_vtime()) {
 164     _vtime_accum = (os::elapsedVTime() - _vtime_start);
 167     } else {
```



```
 168       _vtime_accum = 0.0;
 165   }
 170   }

 167   log_debug(gc, refine)("Stopping %d", _worker_id);
 168 }
______unchanged_portion_omitted_
```

```
**********************************************************
  204096 Thu Jun 30 09:24:12 2016
new/src/share/vm/gc/g1/g1CollectedHeap.cpp
*** NO COMMENTS ***
**********************************************************
______unchanged_portion_omitted_

1793 jint G1CollectedHeap::initialize() {
1794   CollectedHeap::pre_initialize();
1795   os::enable_vtime();

1796   // Necessary to satisfy locking discipline assertions.

1798   MutexLocker x(Heap_lock);

1800   // While there are no constraints in the GC code that HeapWordSize
1801   // be any particular value, there are multiple other areas in the
1802   // system which believe this to be true (e.g. oop->object_size in some
1803   // cases incorrectly returns the size in wordSize units rather than
1804   // HeapWordSize).
1805   guarantee(HeapWordSize == wordSize, "HeapWordSize must equal wordSize");

1807   size_t init_byte_size = collector_policy()->initial_heap_byte_size();
1808   size_t max_byte_size = collector_policy()->max_heap_byte_size();
1809   size_t heap_alignment = collector_policy()->heap_alignment();

1811   // Ensure that the sizes are properly aligned.
1812   Universe::check_alignment(init_byte_size, HeapRegion::GrainBytes, "g1 heap");
1813   Universe::check_alignment(max_byte_size, HeapRegion::GrainBytes, "g1 heap");
1814   Universe::check_alignment(max_byte_size, heap_alignment, "g1 heap");

1816   _refine_cte_cl = new RefineCardTableEntryClosure();

1818   jint ecode = JNI_OK;
1819   _cg1r = ConcurrentG1Refine::create(_refine_cte_cl, &ecode);
1820   if (_cg1r == NULL) {
1821     return ecode;
1822   }

1824   // Reserve the maximum.

1826   // When compressed oops are enabled, the preferred heap base
1827   // is calculated by subtracting the requested size from the
1828   // 32Gb boundary and using the result as the base address for
1829   // heap reservation. If the requested size is not aligned to
1830   // HeapRegion::GrainBytes (i.e. the alignment that is passed
1831   // into the ReservedHeapSpace constructor) then the actual
1832   // base of the reserved heap may end up differing from the
1833   // address that was requested (i.e. the preferred heap base).
1834   // If this happens then we could end up using a non-optimal
1835   // compressed oops mode.

1837   ReservedSpace heap_rs = Universe::reserve_heap(max_byte_size,
1838                                                  heap_alignment);

1840   initialize_reserved_region((HeapWord*)heap_rs.base(), (HeapWord*)(heap_rs.base

1842   // Create the barrier set for the entire reserved region.
1843   G1SATBCardTableLoggingModRefBS* bs
1844     = new G1SATBCardTableLoggingModRefBS(reserved_region());
1845   bs->initialize();
1846   assert(bs->is_a(BarrierSet::G1SATBCTLogging), "sanity");
1847   set_barrier_set(bs);

1849   // Create the hot card cache.
1850   _hot_card_cache = new G1HotCardCache(this);
```



```
1852   // Also create a G1 rem set.
1853   _g1_rem_set = new G1RemSet(this, g1_barrier_set(), _hot_card_cache);

1855   // Carve out the G1 part of the heap.
1856   ReservedSpace g1_rs = heap_rs.first_part(max_byte_size);
1857   size_t page_size = UseLargePages ? os::large_page_size() : os::vm_page_size();
1858   G1RegionToSpaceMapper* heap_storage =
1859     G1RegionToSpaceMapper::create_mapper(g1_rs,
1860                                          g1_rs.size(),
1861                                          page_size,
1862                                          HeapRegion::GrainBytes,
1863                                          1,
1864                                          mtJavaHeap);
1865   os::trace_page_sizes("Heap",
1866                        collector_policy()->min_heap_byte_size(),
1867                        max_byte_size,
1868                        page_size,
1869                        heap_rs.base(),
1870                        heap_rs.size());
1871   heap_storage->set_mapping_changed_listener(&_listener);

1873   // Create storage for the BOT, card table, card counts table (hot card cache)
1874   G1RegionToSpaceMapper* bot_storage =
1875     create_aux_memory_mapper("Block Offset Table",
1876                              G1BlockOffsetTable::compute_size(g1_rs.size() / Hea
1877                              G1BlockOffsetTable::heap_map_factor());

1879   ReservedSpace cardtable_rs(G1SATBCardTableLoggingModRefBS::compute_size(g1_rs.
1880   G1RegionToSpaceMapper* cardtable_storage =
1881     create_aux_memory_mapper("Card Table",
1882                              G1SATBCardTableLoggingModRefBS::compute_size(g1_rs.
1883                              G1SATBCardTableLoggingModRefBS::heap_map_factor());

1885   G1RegionToSpaceMapper* card_counts_storage =
1886     create_aux_memory_mapper("Card Counts Table",
1887                              G1CardCounts::compute_size(g1_rs.size() / HeapWordS
1888                              G1CardCounts::heap_map_factor());

1890   size_t bitmap_size = G1CMBitMap::compute_size(g1_rs.size());
1891   G1RegionToSpaceMapper* prev_bitmap_storage =
1892     create_aux_memory_mapper("Prev Bitmap", bitmap_size, G1CMBitMap::heap_map_fa
1893   G1RegionToSpaceMapper* next_bitmap_storage =
1894     create_aux_memory_mapper("Next Bitmap", bitmap_size, G1CMBitMap::heap_map_fa

1896   _hrm.initialize(heap_storage, prev_bitmap_storage, next_bitmap_storage, bot_st
1897   g1_barrier_set()->initialize(cardtable_storage);
1898   // Do later initialization work for concurrent refinement.
1899   _hot_card_cache->initialize(card_counts_storage);

1901   // 6843694 - ensure that the maximum region index can fit
1902   // in the remembered set structures.
1903   const uint max_region_idx = (1U << (sizeof(RegionIdx_t)*BitsPerByte-1)) - 1;
1904   guarantee((max_regions() - 1) <= max_region_idx, "too many regions");

1906   g1_rem_set()->initialize(max_capacity(), max_regions());

1908   size_t max_cards_per_region = ((size_t)1 << (sizeof(CardIdx_t)*BitsPerByte-1))
1909   guarantee(HeapRegion::CardsPerRegion > 0, "make sure it's initialized");
1910   guarantee(HeapRegion::CardsPerRegion < max_cards_per_region,
1911             "too many cards per region");

1913   FreeRegionList::set_unrealistically_long_length(max_regions() + 1);

1915   _bot = new G1BlockOffsetTable(reserved_region(), bot_storage);
```

```
1917   {
1918     HeapWord* start = _hrm.reserved().start();
1919     HeapWord* end = _hrm.reserved().end();
1920     size_t granularity = HeapRegion::GrainBytes;

1922     _in_cset_fast_test.initialize(start, end, granularity);
1923     _humongous_reclaim_candidates.initialize(start, end, granularity);
1924   }

1926   // Create the G1ConcurrentMark data structure and thread.
1927   // (Must do this late, so that "max_regions" is defined.)
1928   _cm = new G1ConcurrentMark(this, prev_bitmap_storage, next_bitmap_storage);
1929   if (_cm == NULL || !_cm->completed_initialization()) {
1930     vm_shutdown_during_initialization("Could not create/initialize G1ConcurrentM
1931     return JNI_ENOMEM;
1932   }
1933   _cmThread = _cm->cmThread();

1935   // Now expand into the initial heap size.
1936   if (!expand(init_byte_size)) {
1937     vm_shutdown_during_initialization("Failed to allocate initial heap.");
1938     return JNI_ENOMEM;
1939   }

1941   // Perform any initialization actions delegated to the policy.
1942   g1_policy()->init(this, &_collection_set);

1944   JavaThread::satb_mark_queue_set().initialize(SATB_Q_CBL_mon,
1945                                                SATB_Q_FL_lock,
1946                                                G1SATBProcessCompletedThreshold,
1947                                                Shared_SATB_Q_lock);

1949   JavaThread::dirty_card_queue_set().initialize(_refine_cte_cl,
1950                                                 DirtyCardQ_CBL_mon,
1951                                                 DirtyCardQ_FL_lock,
1952                                                 (int)concurrent_g1_refine()->yel
1953                                                 (int)concurrent_g1_refine()->red
1954                                                 Shared_DirtyCardQ_lock,
1955                                                 NULL,  // fl_owner
1956                                                 true); // init_free_ids

1958   dirty_card_queue_set().initialize(NULL, // Should never be called by the Java
1959                                     DirtyCardQ_CBL_mon,
1960                                     DirtyCardQ_FL_lock,
1961                                     -1, // never trigger processing
1962                                     -1, // no limit on length
1963                                     Shared_DirtyCardQ_lock,
1964                                     &JavaThread::dirty_card_queue_set());

1966   // Here we allocate the dummy HeapRegion that is required by the
1967   // G1AllocRegion class.
1968   HeapRegion* dummy_region = _hrm.get_dummy_region();

1970   // We'll re-use the same region whether the alloc region will
1971   // require BOT updates or not and, if it doesn't, then a non-young
1972   // region will complain that it cannot support allocations without
1973   // BOT updates. So we'll tag the dummy region as eden to avoid that.
1974   dummy_region->set_eden();
1975   // Make sure it's full.
1976   dummy_region->set_top(dummy_region->end());
1977   G1AllocRegion::setup(this, dummy_region);

1979   _allocator->init_mutator_alloc_region();

1981   // Do create of the monitoring and management support so that
1982   // values in the heap have been properly initialized.
```



```
1983   _g1mm = new G1MonitoringSupport(this);

1985   G1StringDedup::initialize();

1987   _preserved_marks_set.init(ParallelGCThreads);

1989   return JNI_OK;
1990 }
```

_____***unchanged_portion_omitted_***

****
3655 Thu Jun 30 09:24:12 2016
new/src/share/vm/gc/g1/g1YoungRemSetSamplingThread.cpp
****** NO COMMENTS ******
****
_____***unchanged_portion_omitted_***

```
 53 void G1YoungRemSetSamplingThread::run_service() {
 54   double vtime_start = os::elapsedVTime();

 56   while (!should_terminate()) {
 57     sample_young_list_rs_lengths();

 59     if (os::supports_vtime()) {
 58     _vtime_accum = (os::elapsedVTime() - vtime_start);
 61     } else {
 62       _vtime_accum = 0.0;
 63     }

 59     sleep_before_next_cycle();
 60   }
 61 }
```

_____***unchanged_portion_omitted_***